Encryption and Decryption using Huffman Coding

Submitted in fulfilment for the award in the degree of B. TECH/ INT. MSC.

By

Chandreyi Chowdhury: 19MIY0031

Nikita Anand Joshi: 19BEC0666

Venkateswara Rao Dunne: 19BDS0007

Under guidance of:

Prof. Sanjiban Sekhar Roy

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

<u>**DECLARATION BY THE CANDIDATE**</u>

We hereby declare that the project report entitled "**ENCRYPTION AND DECRYPTION USING HUFFMAN CODING**" submitted by us to VIT University, Vellore is a record of J^{th} component of project work carried out by us under the guidance of **Prof Sanjiban Sekhar Roy**. We further declare that the work reported in this project has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

Place: Vellore
Date: 29th April 2022

Signature of the candidates
Chandreyi Chowdhury: 19MIY0031
Nikita Anand Joshi: 19BEC0666
Venkateswara Rao Dunne: 19BDS0007

Abstract –

Huffman coding is a form of statistical coding. Not all characters in a paragraph occur with the same frequency! Yet all characters are allocated the same amount of space 1 char = 1 byte, be it e or x.
To solve this problem, we need data compression. Huffman coding is a method Proposed by Dr. David A. Huffman in 1952 to compress data.
Back then he mathematically proved that further compression on any character in impossible and his method was the best.
Even tho better compression technics exist in the modern world they only are variations of original Huffman coding.

Hence, we choose this topic to fully understand and try to master this essential part of computer science.

The Basic Algorithm –

1. Scan text to be compressed and tally occurrence of all characters.
2. Sort or prioritize characters based on number of occurrences in text.
3. Build Huffman code tree based on prioritized list.
4. Perform a traversal of tree to determine all code words.
5. Scan text again and create new file using the Huffman codes.

Data Structures Used –

Binary Trees

Features present –

1. Selection of source and destination file of your convenience
2. A Menu like interface to help people unfamiliar with the program operates it
3. Encoding and decoding of your files
4. Works better with larger paragraphs
5. Small run time
6. Dynamic Memory allocation
7. Freeing of allocated memory when program runtime is over to save space
8. Emergency Exit option in case of a mis-click

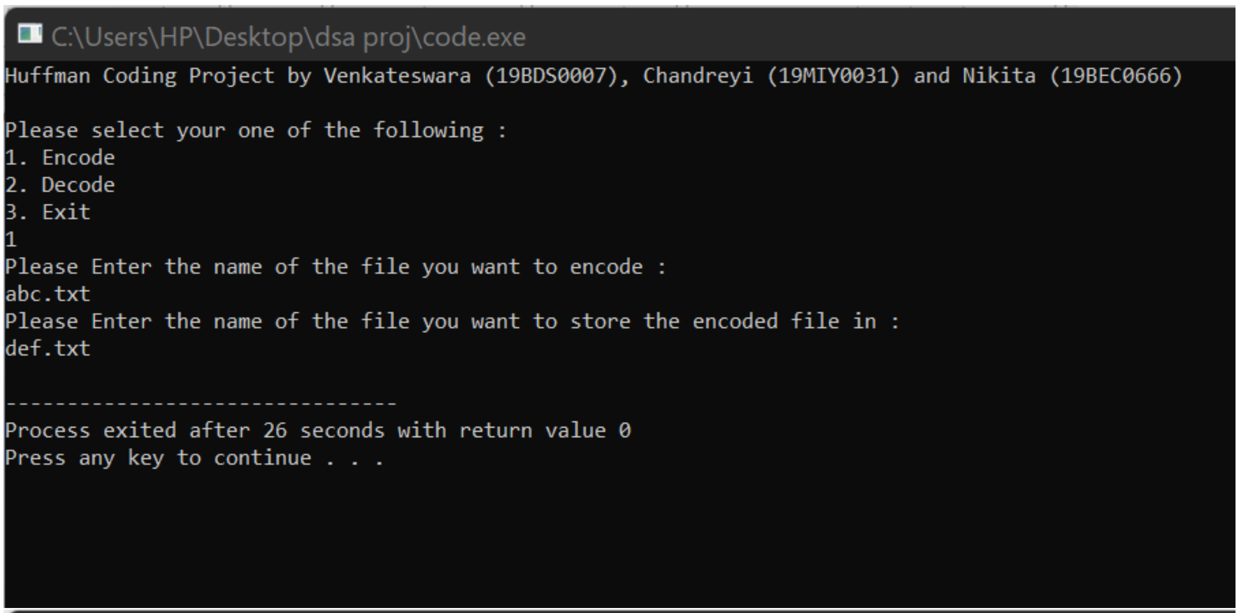
Code –

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_LIMIT 256
// maximum size of the buffer kept 256 cause thats the amount
of english characters
#define FILE_END -1
#define MEM_ALLOC_FAIL -1
int num_alpha = 256;
int num_active = 0;
int *frequen = NULL;
unsigned int orig_size = 0;

typedef struct {
    int index;
    unsigned int weight;
} node_t;

node_t *nodes = NULL;
int num_nodes = 0;
int *leaf_index = NULL;
int *parent_index = NULL;
int free_index = 1;
int *stack;
int stack_top;

unsigned char buffer[MAX_LIMIT];
int bits_in_buffer = 0;
int current_bit = 0;

int eof_input = 0;

int read_header(FILE *f);
int write_header(FILE *f);
int read_bit(FILE *f);
int write_bit(FILE *f, int bit);
int flush_buffer(FILE *f);
void decode_bit_stream(FILE *fin, FILE *fout);
int decode(const char* ifile, const char *ofile);
void encode_alphabet(FILE *fout, int character);
int encode(const char* ifile, const char *ofile);
void build_tree();
void add_leaves();
```

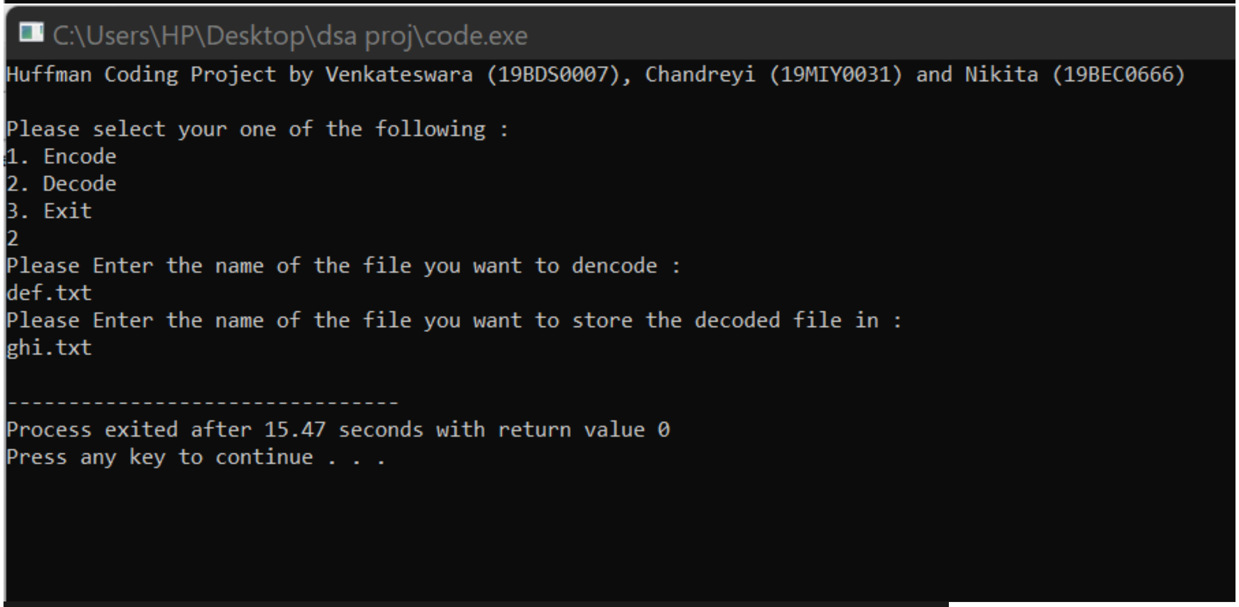
```
int add_node(int index, int weight);
void initialise();

void det_freq(FILE *f) {
    int c;
    while ((c = fgetc(f)) != EOF) {
        ++frequen[c];
        ++orig_size;
    }
    for (c = 0; c < num_alpha; ++c)
        if (frequen[c] > 0)
            ++num_active;
}

void initialise() {
    frequen = (int *)calloc(2 * num_alpha, sizeof(int));
    leaf_index = frequen + num_alpha - 1;
}

void allocate_tree() {
    nodes = (node_t *)calloc(2 * num_active, sizeof(node_t));
    parent_index = (int *)calloc(num_active, sizeof(int));
}


int add_node(int index, int weight) {
    int i = num_nodes++;
    while (i > 0 && nodes[i].weight > weight) {
        memcpy(&nodes[i + 1], &nodes[i], sizeof(node_t));
        if (nodes[i].index < 0)
            ++leaf_index[-nodes[i].index];
        else
            ++parent_index[nodes[i].index];
        --i;
    }

    ++i;
    nodes[i].index = index;
    nodes[i].weight = weight;
    if (index < 0)
        leaf_index[-index] = i;
    else
        parent_index[index] = i;
    return i;
}

void add_leaves() {
    int i, freq;
    for (i = 0; i < num_alpha; ++i) {
        freq = frequen[i];
        if (freq > 0)
            add_node(-(i + 1), freq);
    }
}

void build_tree() {
    int a, b, index;
    while (free_index < num_nodes) {
        a = free_index++;
        b = free_index++;
        index = add_node(b/2,
            nodes[a].weight + nodes[b].weight);
        parent_index[b/2] = index;
    }
}


int encode(const char* ifile, const char *ofile) {
    FILE *fin, *fout;
    fin = fopen(ifile, "rb");
    fout = fopen(ofile, "wb");

    det_freq(fin);
    stack = (int *) calloc(num_active - 1, sizeof(int));
    allocate_tree();

    add_leaves();
    write_header(fout);
    build_tree();
    fseek(fin, 0, SEEK_SET);
    int c;
    while ((c = fgetc(fin)) != EOF)
        encode_alphabet(fout, c);
    flush_buffer(fout);
    free(stack);
    fclose(fin);
    fclose(fout);

    return 0;
}

void encode_alphabet(FILE *fout, int character) {
    int node_index;
    stack_top = 0;
    node_index = leaf_index[character + 1];
    while (node_index < num_nodes) {
        stack[stack_top++] = node_index % 2;
        node_index = parent_index[(node_index + 1) / 2];
    }
    while (--stack_top > -1)
        write_bit(fout, stack[stack_top]);
}

int decode(const char* ifile, const char *ofile) {
    FILE *fin, *fout;
    fin = fopen(ifile, "rb");
    fout = fopen(ofile, "wb");

    if (read_header(fin) == 0) {
        build_tree();
        decode_bit_stream(fin, fout);
    }
    fclose(fin);
    fclose(fout);

    return 0;
}

void decode_bit_stream(FILE *fin, FILE *fout) {
    int i = 0, bit, node_index = nodes[num_nodes].index;
    while (1) {
        bit = read_bit(fin);
        if (bit == -1)
            break;
        node_index = nodes[node_index * 2 - bit].index;
        if (node_index < 0) {
            char c = -node_index - 1;
            fwrite(&c, 1, 1, fout);
            if (++i == orig_size)
                break;
            node_index = nodes[num_nodes].index;
        }
    }
}

int write_bit(FILE *f, int bit) {
    if (bits_in_buffer == MAX_LIMIT << 3) {
        size_t bytes_written =
            fwrite(buffer, 1, MAX_LIMIT, f);
        bits_in_buffer = 0;
        memset(buffer, 0, MAX_LIMIT);
    }
    if (bit)
        buffer[bits_in_buffer >> 3] |=
            (0x1 << (7 - bits_in_buffer % 8));
    ++bits_in_buffer;
    return 0;
}

int flush_buffer(FILE *f) {
    if (bits_in_buffer) {
        size_t bytes_written =
            fwrite(buffer, 1,
                (bits_in_buffer + 7) >> 3, f);
        if (bytes_written < MAX_LIMIT && ferror(f))
            return -1;
        bits_in_buffer = 0;
    }
    return 0;
}

int read_bit(FILE *f) {
    if (current_bit == bits_in_buffer) {
        if (eof_input)
            return FILE_END;
        else {
            size_t bytes_read =
                fread(buffer, 1, MAX_LIMIT, f);
            if (bytes_read < MAX_LIMIT) {
                if (feof(f))
                    eof_input = 1;
            }
            bits_in_buffer = bytes_read << 3;
            current_bit = 0;
        }
    }

    if (bits_in_buffer == 0)
        return FILE_END;
    int bit = (buffer[current_bit >> 3] >>
        (7 - current_bit % 8)) & 0x1;
    ++current_bit;
    return bit;
}

int write_header(FILE *f) {
     int i, j, byte = 0,
         size = sizeof(unsigned int) + 1 + num_active * (1 +
sizeof(int));
     unsigned int weight;
     char *buffer = (char *) calloc(size, 1);
     if (buffer == NULL)
```

```
            return MEM_ALLOC_FAIL;

        j = sizeof(int);
        while (j--)
            buffer[byte++] =
                (orig_size >> (j << 3)) & 0xff;
        buffer[byte++] = (char) num_active;
        for (i = 1; i <= num_active; ++i) {
            weight = nodes[i].weight;
            buffer[byte++] =
                (char) (-nodes[i].index - 1);
            j = sizeof(int);
            while (j--)
                buffer[byte++] =
                    (weight >> (j << 3)) & 0xff;
        }
        fwrite(buffer, 1, size, f);
        free(buffer);
        return 0;
    }

    int read_header(FILE *f) {
        int i, j, byte = 0, size;
        size_t bytes_read;
        unsigned char buff[4];

        bytes_read = fread(&buff, 1, sizeof(int), f);
        if (bytes_read < 1)
            return FILE_END;
        byte = 0;
        orig_size = buff[byte++];
        while (byte < sizeof(int))
            orig_size = (orig_size << (1 << 3)) | buff[byte++];

        bytes_read = fread(&num_active, 1, 1, f);
        if (bytes_read < 1)
            return FILE_END;

        allocate_tree();
        size = num_active * (1 + sizeof(int));
        unsigned int weight;
        char *buffer = (char *) calloc(size, 1);
        if (buffer == NULL)
            return MEM_ALLOC_FAIL;
        fread(buffer, 1, size, f);
        byte = 0;
        for  (i  =  1;  i  <=  num_active;  ++i)  {
            nodes[i].index = -(buffer[byte++] + 1);
            j = 0;
            weight = (unsigned char) buffer[byte++];
            while (++j < sizeof(int)) {
                weight = (weight << (1 << 3)) |
                    (unsigned char) buffer[byte++];
            }
            nodes[i].weight = weight;
        }
        num_nodes = (int) num_active;
        free(buffer);
        return 0;
   }



   int main() {
       initialise();
       int loop_var = 0;
        printf("Huffman Coding Project by Chandreyi Chowdhury
        (19MIY0031), Nikita Anand Joshi (19BEC0714),
        Venkateswara Rao Dunne (19BDS0007) \n\n");
        printf("Please select your one of the following :\n");
        printf("1. Encode \n2. Decode \n3. Exit\n");
        scanf("%d" , &loop_var);
       if (loop_var == 1){
        char source[40] , destination[40];
        printf("Please Enter the name of the file you want to
   encode :\n");
        scanf("%s" , &source);
        printf("Please Enter the name of the file you want to
   store the encoded file in :\n");
        scanf("%s" , &destination);
             encode(source, destination);
        }else if (loop_var == 2){
           char source[40] , destination[40];
        printf("Please Enter the name of the file you want to
   dencode :\n");
        scanf("%s" , &source);
        printf("Please Enter the name of the file you want to
   store the decoded file in :\n");
        scanf("%s" , &destination);
             decode(source, destination);
        }

       free(parent_index);
       free(frequen);
       free(nodes);
       return 0;
   }
```

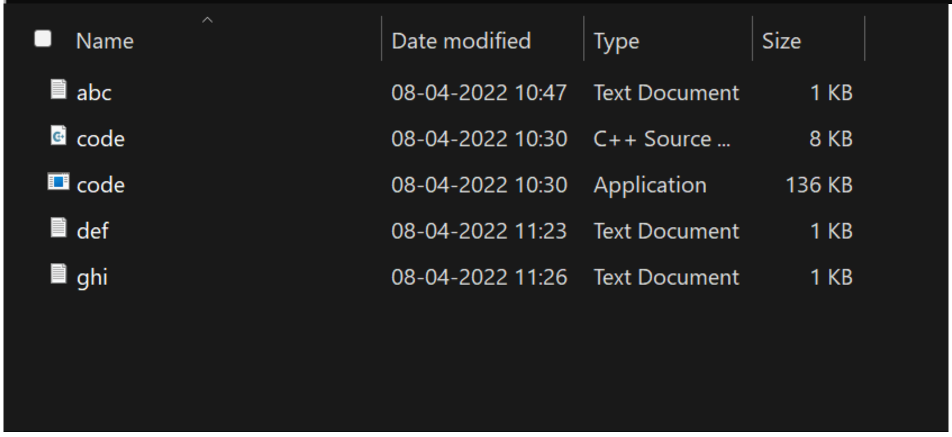
Screen Shots –



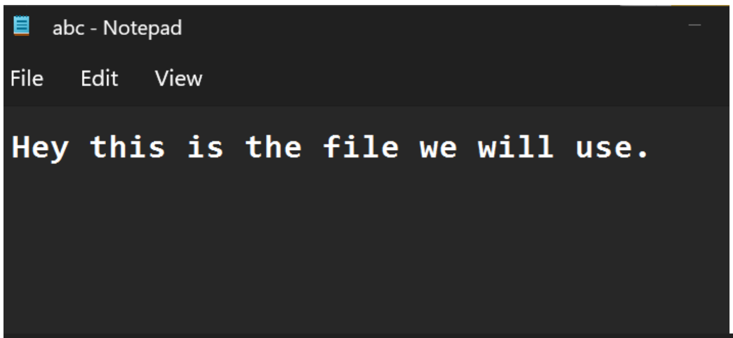
```
Huffman Coding Project by Venkateswara (19BDS0007), Chandreyi (19MIY0031) and Nikita (19BEC0666)

Please select your one of the following :
1. Encode
2. Decode
3. Exit
1
Please Enter the name of the file you want to encode :
abc.txt
Please Enter the name of the file you want to store the encoded file in :
def.txt

--------------------------------
Process exited after 26 seconds with return value 0
Press any key to continue . . .
```



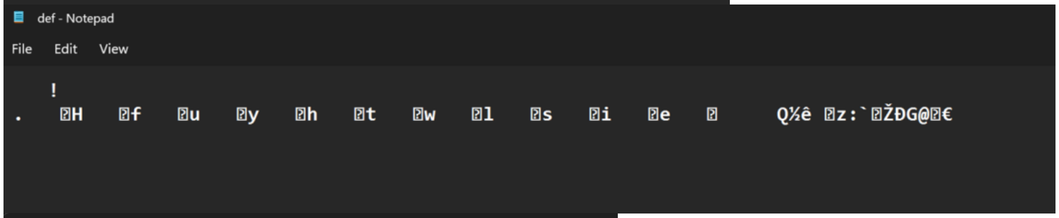
```
Huffman Coding Project by Venkateswara (19BDS0007), Chandreyi (19MIY0031) and Nikita (19BEC0666)

Please select your one of the following :
1. Encode
2. Decode
3. Exit
2
Please Enter the name of the file you want to dencode :
def.txt
Please Enter the name of the file you want to store the decoded file in :
ghi.txt

--------------------------------
Process exited after 15.47 seconds with return value 0
Press any key to continue . . .
```



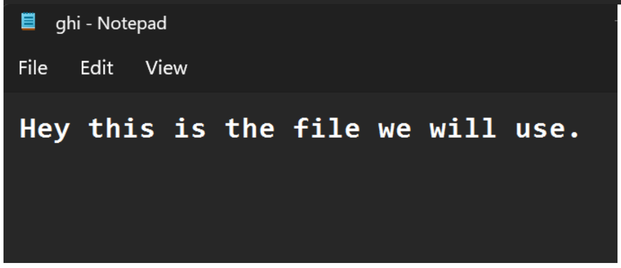
Name	Date modified	Type	Size
abc	08-04-2022 10:47	Text Document	1 KB
code	08-04-2022 10:30	C++ Source ...	8 KB
code	08-04-2022 10:30	Application	136 KB
def	08-04-2022 11:23	Text Document	1 KB
ghi	08-04-2022 11:26	Text Document	1 KB



```
Hey this is the file we will use.
```

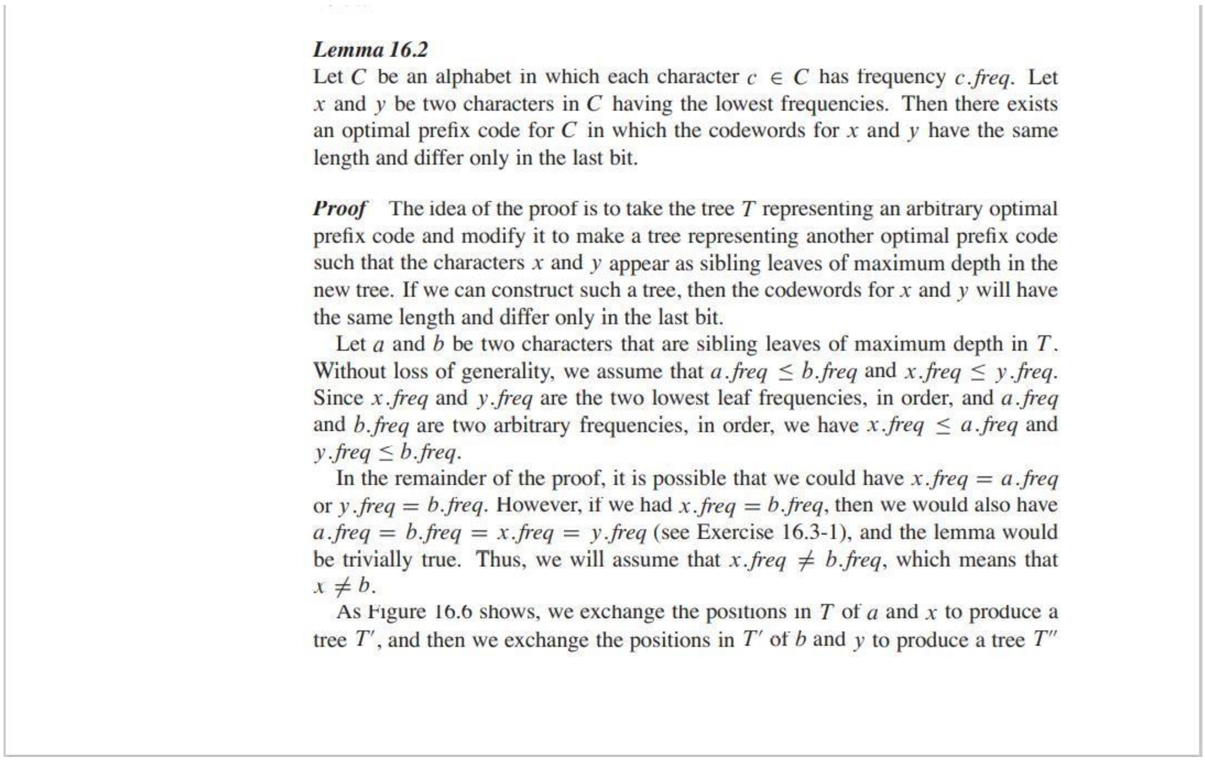


```
    !
.   ⍰H   ⍰f   ⍰u   ⍰y   ⍰h   ⍰t   ⍰w   ⍰l   ⍰s   ⍰i   ⍰e   ⍰     Q½ê ⍰z:`⍰ŽÐG@⍰€
```



```
Hey this is the file we will use.
```

Lemmas –



Lemma 16.2
Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

Proof The idea of the proof is to take the tree T representing an arbitrary optimal prefix code and modify it to make a tree representing another optimal prefix code such that the characters x and y appear as sibling leaves of maximum depth in the new tree. If we can construct such a tree, then the codewords for x and y will have the same length and differ only in the last bit.

Let a and b be two characters that are sibling leaves of maximum depth in T. Without loss of generality, we assume that $a.freq \leq b.freq$ and $x.freq \leq y.freq$. Since $x.freq$ and $y.freq$ are the two lowest leaf frequencies, in order, and $a.freq$ and $b.freq$ are two arbitrary frequencies, in order, we have $x.freq \leq a.freq$ and $y.freq \leq b.freq$.

In the remainder of the proof, it is possible that we could have $x.freq = a.freq$ or $y.freq = b.freq$. However, if we had $x.freq = b.freq$, then we would also have $a.freq = b.freq = x.freq = y.freq$ (see Exercise 16.3-1), and the lemma would be trivially true. Thus, we will assume that $x.freq \neq b.freq$, which means that $x \neq b$.

As Figure 16.6 shows, we exchange the positions in T of a and x to produce a tree T', and then we exchange the positions in T' of b and y to produce a tree T''

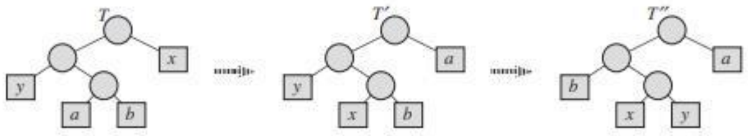


Figure 16.6 An illustration of the key step in the proof of Lemma 16.2. In the optimal tree T, leaves a and b are two siblings of maximum depth. Leaves x and y are the two characters with the lowest frequencies; they appear in arbitrary positions in T. Assuming that $x \neq b$, swapping leaves a and x produces tree T', and then swapping leaves b and y produces tree T''. Since each swap does not increase the cost, the resulting tree T'' is also an optimal tree.

in which x and y are sibling leaves of maximum depth. (Note that if $x = b$ but $y \neq a$, then tree T'' does not have x and y as sibling leaves of maximum depth. Because we assume that $x \neq b$, this situation cannot occur.) By equation (16.4), the difference in cost between T and T' is

$$\begin{aligned}
B(T) - B(T') & \\
&= \sum_{c \in C} c.\mathit{freq} \cdot d_T(c) - \sum_{c \in C} c.\mathit{freq} \cdot d_{T'}(c) \\
&= x.\mathit{freq} \cdot d_T(x) + a.\mathit{freq} \cdot d_T(a) - x.\mathit{freq} \cdot d_{T'}(x) - a.\mathit{freq} \cdot d_{T'}(a) \\
&= x.\mathit{freq} \cdot d_T(x) + a.\mathit{freq} \cdot d_T(a) - x.\mathit{freq} \cdot d_T(a) - a.\mathit{freq} \cdot d_T(x) \\
&= (a.\mathit{freq} - x.\mathit{freq})(d_T(a) - d_T(x)) \\
&\geq 0 ,
\end{aligned}$$

because both $a.\mathit{freq} - x.\mathit{freq}$ and $d_T(a) - d_T(x)$ are nonnegative. More specifically, $a.\mathit{freq} - x.\mathit{freq}$ is nonnegative because x is a minimum-frequency leaf, and $d_T(a) - d_T(x)$ is nonnegative because a is a leaf of maximum depth in T. Similarly, exchanging y and b does not increase the cost, and so $B(T') - B(T'')$ is nonnegative. Therefore, $B(T'') \leq B(T)$, and since T is optimal, we have $B(T) \leq B(T'')$, which implies $B(T'') = B(T)$. Thus, T'' is an optimal tree in which x and y appear as sibling leaves of maximum depth, from which the lemma follows. ■

Lemma 16.2 implies that the process of building up an optimal tree by mergers can, without loss of generality, begin with the greedy choice of merging together those two characters of lowest frequency. Why is this a greedy choice? We can view the cost of a single merger as being the sum of the frequencies of the two items being merged. Exercise 16.3-4 shows that the total cost of the tree constructed equals the sum of the costs of its mergers. Of all possible mergers at each step, HUFFMAN chooses the one that incurs the least cost.

The next lemma shows that the problem of constructing optimal prefix codes has the optimal-substructure property.

Lemma 16.3
Let C be a given alphabet with frequency $c.\mathit{freq}$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with the characters x and y removed and a new character z added, so that $C' = C - \{x, y\} \cup \{z\}$. Define f for C' as for C, except that $z.\mathit{freq} = x.\mathit{freq} + y.\mathit{freq}$. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

Proof We first show how to express the cost $B(T)$ of tree T in terms of the cost $B(T')$ of tree T', by considering the component costs in equation (16.4). For each character $c \in C - \{x, y\}$, we have that $d_T(c) = d_{T'}(c)$, and hence $c.\mathit{freq} \cdot d_T(c) = c.\mathit{freq} \cdot d_{T'}(c)$. Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

$$\begin{aligned} x.\mathit{freq} \cdot d_T(x) + y.\mathit{freq} \cdot d_T(y) &= (x.\mathit{freq} + y.\mathit{freq})(d_{T'}(z) + 1) \\ &= z.\mathit{freq} \cdot d_{T'}(z) + (x.\mathit{freq} + y.\mathit{freq}), \end{aligned}$$

from which we conclude that

$$B(T) = B(T') + x.\mathit{freq} + y.\mathit{freq}$$

or, equivalently,

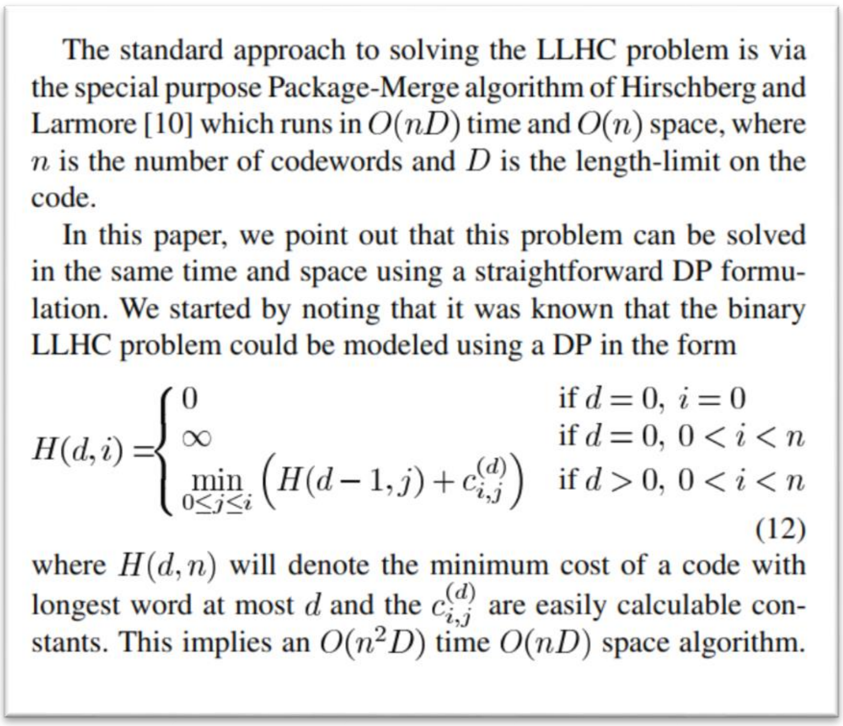
$$B(T') = B(T) - x.\mathit{freq} - y.\mathit{freq}.$$

We now prove the lemma by contradiction. Suppose that T does not represent an optimal prefix code for C. Then there exists an optimal tree T'' such that $B(T'') < B(T)$. Without loss of generality (by Lemma 16.2), T'' has x and y as siblings. Let T''' be the tree T'' with the common parent of x and y replaced by a leaf z with frequency $z.\mathit{freq} = x.\mathit{freq} + y.\mathit{freq}$. Then

$$\begin{aligned} B(T''') &= B(T'') - x.\mathit{freq} - y.\mathit{freq} \\ &< B(T) - x.\mathit{freq} - y.\mathit{freq} \\ &= B(T'), \end{aligned}$$

yielding a contradiction to the assumption that T' represents an optimal prefix code for C'. Thus, T must represent an optimal prefix code for the alphabet C. ■

Furthur: Dynamic Approach

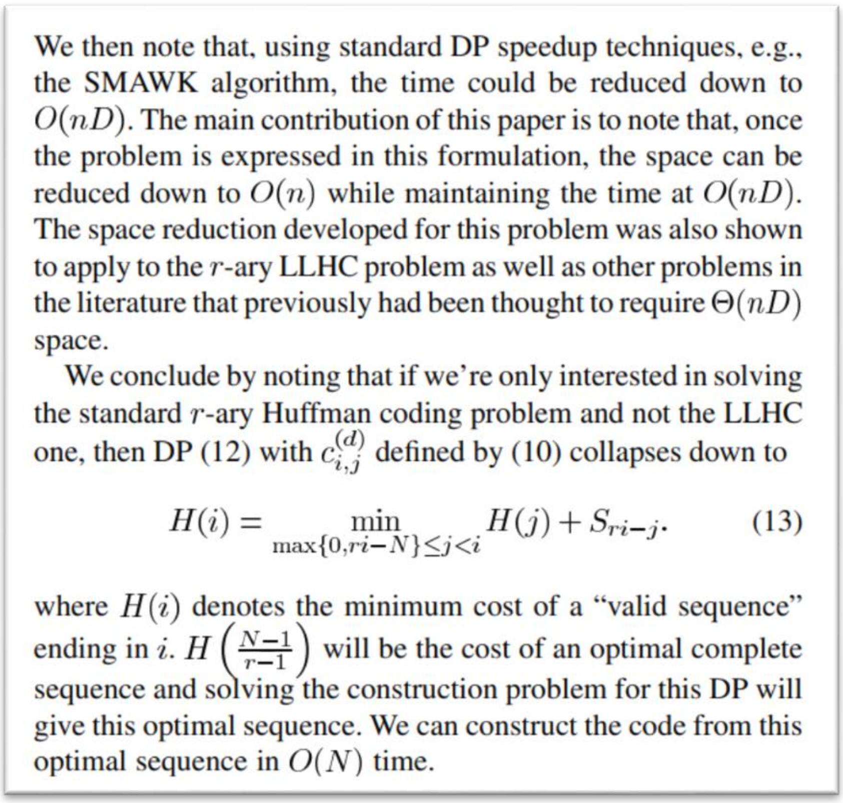


The standard approach to solving the LLHC problem is via the special purpose Package-Merge algorithm of Hirschberg and Larmore [10] which runs in $O(nD)$ time and $O(n)$ space, where n is the number of codewords and D is the length-limit on the code.

In this paper, we point out that this problem can be solved in the same time and space using a straightforward DP formulation. We started by noting that it was known that the binary LLHC problem could be modeled using a DP in the form

$$H(d,i) = \begin{cases} 0 & \text{if } d=0,\ i=0 \\ \infty & \text{if } d=0,\ 0<i<n \\ \min\limits_{0\le j\le i} \left(H(d-1,j) + c_{i,j}^{(d)} \right) & \text{if } d>0,\ 0<i<n \end{cases} \tag{12}$$

where $H(d,n)$ will denote the minimum cost of a code with longest word at most d and the $c_{i,j}^{(d)}$ are easily calculable constants. This implies an $O(n^2D)$ time $O(nD)$ space algorithm.



We then note that, using standard DP speedup techniques, e.g., the SMAWK algorithm, the time could be reduced down to $O(nD)$. The main contribution of this paper is to note that, once the problem is expressed in this formulation, the space can be reduced down to $O(n)$ while maintaining the time at $O(nD)$. The space reduction developed for this problem was also shown to apply to the r-ary LLHC problem as well as other problems in the literature that previously had been thought to require $\Theta(nD)$ space.

We conclude by noting that if we're only interested in solving the standard r-ary Huffman coding problem and not the LLHC one, then DP (12) with $c_{i,j}^{(d)}$ defined by (10) collapses down to

$$H(i) = \min_{\max\{0, ri-N\}\le j<i} H(j) + S_{ri-j}. \tag{13}$$

where $H(i)$ denotes the minimum cost of a "valid sequence" ending in i. $H\left(\frac{N-1}{r-1}\right)$ will be the cost of an optimal complete sequence and solving the construction problem for this DP will give this optimal sequence. We can construct the code from this optimal sequence in $O(N)$ time.

To summarize, by transforming -ary Huffman coding into a DP and using sophisticated tools, one can solve the problem in time. This is not of practical interest, though, since the simple, greedy, Huffman encoding algorithm is just as fast. Where the DP formulation helps is in the length-limited Huffman coding (LLHC) problem, exactly where the greedy procedure fails. In that case they have the added practical benefit of being able to use the simple SMAWK algorithm. (The SMAWK algorithm is an algorithm for finding the minimum value in each row of an implicitly-defined totally monotone matrix. It is named after the initials of its five inventors, Peter Shor, Shlomo Moran, Alok Aggarwal, Robert Wilber, and Maria Klawe.)

References –

Research paper for DP approach:
https://ieeexplore.ieee.org/document/5508613
Stackoverflow - Fixing file handling errors
Github - Viewing sample code to understand the concept
Introduction to Algorithms – Thomas Corman